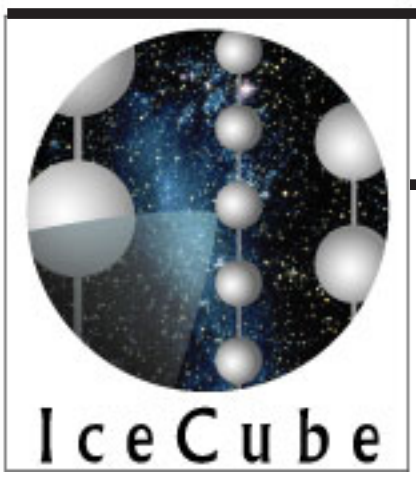

IceCube Software

IceCube Software Configuration Management Plan

Simon Patton L.B.N.L.

This document provides a description of the IceCube software configuration management plan. The aim of this plan is to provide traceability of the IceCube software so that it is possible to reproduce any software that has been used in conjunction with any part of the IceCube “production” data.

Last Modified: February 26, 2004 3:58 pm **Version:**0.0

1.0 Introduction

1.1 Purpose

The purpose of this document is to describe the concepts and tools that are used to manage the configuration of “production” software with the IceCube experiment. The definition of “production” software is documented elsewhere [SDD].

1.2 Scope

This document covers all the issues that relate to configuration of IceCube “production” software. These issues include:

- What are configuration items within the software configuration management plan.
- What tools are used to manage these configuration items.
- How are version numbers assigned to these configuration items.
- How is software built and released within the software configuration management plan.

1.3 Definitions, acronyms and abbreviations

project: A set of source and resource files that are used to create zero or more binaries along with a library, a jarfile or a library-jarfile pair.

resource file: A file that needs no processing to be ready for inclusion in the final product of a project.

SCM: Software Configuration Management.

source file: A file that needs to be processed, usually by a compiler or similar application, so that the resulting output files can be included in the final product of a project.

tag: A string that identifies a unique set of file and associated version that make up a project.

1.4 References

The following references are used throughout this document.

[SDD] The Software Design Description for Icecube “Production” Software.

1.5 Overview of this document

This document starts by giving a quick outline of what is software configuration management and why it is necessary for the IceCube experiment. It goes on to describe what are the basic units within the software configuration management plan and how these can be tracked. Next there is an explanation of how these basic units can be grouped into larger ones so that entire software releases can be tracked by a single version number. Once this groundwork has been established the mechanism for building known software

within the software configuration management plan is described, followed by a description of how software built by such a mechanism can become a Software Release.

2.0 The Benefits of SCM

The key to any credible scientific measurement is reproducibility. If the measurement can not be recreated from its original inputs then it is not possible for other people to verify that the measurement is a genuine scientific result and not simply an artifact of the software with which it was processed and thus can not be defined as credible.This means that all scientific endeavours, not just IceCube, must be able to tell which software was run on which data. The aim of Software configuration Management (SCM) is to provide the means by which this data-software relationship can be traced.

3.0 The Basic Unit

The basic unit that comes under software configuration control is a project.A project is a set of source and resource files that are used to create zero or more binaries along with a library, a jarfile or a library-jarfile pair. The contents of a project should be conceptually related to that they implement the solution to a single issue.

As a project is the smallest unit under configuration control it is also the standard unit that is used when accessing the code repository. For example the following two command should be used to checkout the contents of the `preston` project from and archive that contents back into the code repository:

```
[patton@glacier]$ bfd checkout preston
[patton@glacier]$ bfd archive preston
```

3.1 Tracking the contents of a project

The whole aim of SCM is to know which code has been applied to a given data. Therefore it is necessary to be able to tag any version of a project that has been applied to data. The only requirement for this tag is that it be unique within the context of each given project and thus unambiguously traceable.

Having stated this, it is helpfully to define two different aspects of a project: its public interface; and its private implementation (See 4.1 ‘‘Nesting Composite Units’’ to find out why this distinction is necessary). The public interface is composed of those function and method signatures that can be used by code outside the project, while the private implementation is everything else. Changes in these two aspect between two version of a project can be captured in the tag used to identify each version of the project. This then makes it easier to make statement about how different versions of the project will operate with respect to other projects that depend on this project.

All “production” versions of IceCube software must have a tag of the following form:

```
VXX-YY-ZZ
```

V: This indicates that the information that follows it is related to a version tag as opposed to a branch tag[1].

XX: This is the major feature number. A difference in this number between two versions of the project imply there has been a major change in its public interface and there is no expectation of interchangeability between these two versions.

YY: This is the minor feature number. Between two version of a project, both of which have the same major feature number, the version with the higher minor feature number can be used in place of the version with the lower feature number. The practical result of this is that as the minor version number increases the project's public interface can increase in content, but nothing can be removed - that would require a change in the major feature number.

ZZ: This is the bug fix number. Between two version of a project, both of which have the same major and minor feature numbers, either version can be used in place of the other.

"Production" tags are generated whenever a project is "delivered". A developer uses the act of delivery to state that this version of the project has a known set of features[1], will pass all of its unit tests and can be used in conjunction with the rest of IceCube's software. The following is a example of delivering the `preston` project after a modification has added something to its public interface, i.e. the modification requires a change in the minor feature number:

```
[patton@glacier]$ bfd deliver -n preston
```

The `bfd deliver` command has options to indicate whether the modification being delivered requires a change in the major feature number (`-j`), the minor feature number (`-n`) or the bug fix number (`-b`).

4.0 Composite Units

It is expected that the final IceCube software will be composed of a number of different projects. Some of these projects are more closely related to each other than others. In any case, trying to keep track of a whole range of projects and the version tags would not be easy without additional tools. One of these additional tools is the meta-project.

A meta-project is used to capture information about a set of projects and their versions that all work together. For example the `DOM-MB` meta-project contains a list of all of the projects necessary to create the software that executes on a DOM mainboard. Any particular version of the `DOM-MB` project will contain a set of project-version pairs that defines an exact and complete set of files that make up that version of the `DOM-MB` project.

As well as defining a set of project versions that function together, a meta-project can act like a regular project, i.e. it can produce binaries along with a library, a jarfile or a library-jarfile pair. This behavior is useful if the product of the meta-project requires the contained projects in order to be created. From example the hex file that is the product of the `DOM-MB` project requires the products of its contained projects in order it to be built.

1. Branching has yet to be addressed in detail!

1. The mechanism for documenting features is still under development.

4.1 Nesting Composite Units

As meta-projects are designed to be behave identically with respect to configuration management as projects, there is no reason that one meta-project can not contain another one. However, this does raise a number of issues.

Firstly, cyclic inclusion of meta-project, i.e. putting into one meta-project another meta-project which contains the first one, is, at best, difficult to manage if not impossible. As the benefit of this particular pattern appear to be minimal it is not allowed[1].

A more critical issue is when a meta-project contains two or more references, either directly or via contained meta-projects, to different versions of the same project. In this case the following rules are used to determine which version, if any, of the project is used:

- The meta-project is parsed to find all references to the contained project.
- If all the versions of the project do not have the same major feature number then the meta-project is considered invalid[2].
- If all the versions of the project do not have the same minor feature number, only those with the highest minor feature number are considered in the next step, otherwise they are all considered.
- The version of the project with the highest bug fix number is the version of the project that is considered as being contained in the original meta-project.

5.0 Building software products

To ensure the reproducibility of software it is important to minimize the possibility of error when building software products. To this end, the creation of the set of products from any given project or meta-project is done with a single command. the `bfd produce` command. The following is an example of this command used to create the products of the `preston` project.

```
[patton@glacier]$ bfd produce -r V00-01-01 preston \
                                        /home/icecube/tools linux-i386
```

The `-r` option is used to specify the version of the project which should be checked out from the repository, while the first argument is the name of the project to be built. Any remaining arguments are passed on as the arguments to the `bfd init` command which is executed as part of this command.

The `bfd produce` command does the following:

- generate a unique build number;
- create a new workspace (whose name is based on the build number);
- initialize that workspace;
- checkout the specified version of the project;

1. `bfd` needs to be extended to catch and forbid this configuration.
2. `bfd` needs to be extended to catch and forbid this configuration.

- build the products that can be generated for all of the projects in the workspace;
- create a tar-ball of all the products that have been generated.

It should be noted that each build has a unique number and therefore this number can be used to determine the exact environment and instance that existed at the time of the build. Thus the build number allows for complete traceability of the code.

6.0 Releasing software products

The release of a set of software products means that these products are now available to be executed as part of IceCube's "production" software. The process of releasing a set of products entails taking a given build of those products (See 5.0 ''Building software products''), certifying that it has the feature claimed and noting any deficiencies it may contain. Once the build is certified it is "released" by placing links from the release area on glacier to the workspace that was produced by the build[1].

7.0 Tracing software

There is not point in having a SCM system, unless there is a way to access the information from the final software products. This section outlines how that can, or will be, done.

7.1 Jarfiles

The version number of each java package in a jarfile, which is set to match that of the project in which is was contained, and the build number are sorted in the jarfile's manifest. This can then be accessed either programatically by Java code (See the `java.lang.Pacakage` class API for details), or by inspecting the contents of the `MANIFEST.MF` file in the jarfile itself.

7.2 C/C++ files

No uniform policy exists at present for how versions can be tracked in C/C++ files. Currently the version number of the project along with the build number are available via the `cpp` macros `PROJECT_TAG` and ICESOFT_BUILD respectively. However there is not consensus on how that should be used.

1. This is currently done by hand, but should be automated.